Object Database Standards, Languages, and Design

Chapter 21
Announcement

- HW 3:
  - 10%
  - Due 2\textsuperscript{nd} of June.

- Quiz 3
  - 3%
  - On Saturday May 11
  - Chapter 21
Chapter Objectives

- Discuss the importance of standards (e.g., portability, interoperability)

- Introduce Object Data Management Group (ODMG):

- Present Object Database Conceptual Design
Chapter Outline

- Advantages of Standards
- Overview of the Object Model ODMG
- The Object Definition Language DDL
- The Object Query Language OQL
- Object Database Conceptual Model
One of the main reasons for the success of RDBMS is the SQL standard.

Standards are essential for:
- Portability (ability to be executed in different systems) and
- Interoperability (ability to access multiple systems)

As a result a consortium of ODBMS vendors formed a standard known as ODMG (Object Data Management Group)
The ODMG standard is made up of several parts:

- Object module
- Object definition Language (ODL)
- Object Query Language (OQL)
- Bindings to O-O programming languages (OOPLs)
ODMG Object Model

- ODMG object model:
  - Is the data model upon which the ODL and the OQL are based
  - Provides the data types, type constructors, and other concepts that can be utilized in the ODL to specify object database schemas
  - Provide a standard terminology
ODMG Basic Building Blocks

The basic building blocks of the object model are

- Objects
- Literlas

An object has four characteristics

1. Identifier: unique system-wide identifier
2. Name: unique within a particular database and/or program; it is optional
3. Lifetime: persistent vs transient
4. Structure: specifies how object is constructed by the type constructor and whether it is a collection or an *atomic* object
ODMG Literals

A literal has a current value but not an identifier

Three types of literals

1. Atomic literal: predefined; basic data type values (e.g., short, float, boolean, char)

2. Structured: values that are constructed by tuple constructors. (e.g., Date, Time, Interval, Timestamp, etc)

3. Collection: a collection (e.g., set, list, array, bag, dictionary) of values or objects
ODMG Interface and Class Definition

- ODMG supports two concepts for specifying object types:
  - Interface
  - Class

- There are similarities and differences between interfaces and classes

- Both have behaviors (operations) and state (attributes and relationships)
An interface is a specification of the abstract behavior of an object type.

State properties of an interface (i.e., its attributes and relationships) cannot be inherited from.

Objects cannot be instantiated from an interface.

There are many built-in object interfaces (e.g., Object, Date, Time, Collection, Array, List);
ODMG Interface Definition

- **Example**

  ```
  interface Object {
      ...
      boolean same_as(in object other_object);
      Object copy();
      Void delete();
  }
  ```

- **Note**: `interface` is ODMG’s keyword for class/type
Built-in Interfaces for Date Objects

Example

```java
interface Date: Object {
    enum weekday { sun, mon, tue, wed, thu, fri, sat; }
    enum Month { jan, feb, mar, ..., dec; }
    unsigned short year();
    unsigned short month();
    unsigned short day();
    ...
    boolean is_equal(in Date other_date);
};
```
Built-in Interfaces for Collection Objects

- A collection object inherits the basic collection interface, for example:
  - `cardinality()`
  - `is_empty()`
  - `insert_element()`
  - `remove_element()`
  - `contains_element()`
  - `create_iterator()`
Collection Types

- Collection objects are further specialized into types like a set, list, bag, array, and dictionary.

- Each collection type may provide additional interfaces, for example, a set provides:
  - `create_union()`
  - `create_difference()`
  - `is_subst_of()`
  - `is_superset_of()`
  - `is_proper_subset_of()`
Object Inheritance Hierarchy

Built-in interfaces of the object module
ODMG Class

- A class is a specification of abstract behavior and state of an object type

- A class is Instantiable

- Supports “extends” inheritance to allow both state and behavior inheritance among classes. Unlike interface in which only behavior is inherited.

- Multiple inheritance via “extends” is not allowed
Atomic Objects

- Atomic objects are user-defined objects and are defined via keyword **class**

- An example:

```java
class Employee (extent all_employees key ssn) {
    attribute string name;
    attribute string ssn;
    attribute short age;
    relationship Dept works_for;
    void reassign(in string new_name);
}
```
Class Extents

- An ODMG object can have an extent defined via a class declaration.

- Each extent is given a name and will contain all persistent objects of that class.

- For Employee class, for example, the extent is called all_employees.

- This is similar to creating an object of type `Set<Employee>` and making it persistent.
A class key consists of one or more unique attributes.

For the Employee class, the key is ssn. Thus each employee is expected to have a unique ssn.

Keys can be composite, e.g.,
(key dnumber, dname)
Object Factory

- An object factory is used to generate individual objects via its operations.

- An example:

```java
interface ObjectFactory {
    Object new ();
};
```

- `new()` returns new objects with an `object_id`.

- One can create their own factory interface by inheriting the above interface.
Object Definition Language (ODL)

- ODL supports semantics constructs of ODMG
- ODL is independent of any programming language
- ODL is used to create object specification (classes and interfaces)
- ODL is not used for database manipulation, OQL is.
Graphical notation for representing ODL schemas

- **Interface**
  - Person-IF
- **Class**
  - Student

**Object specification**

- **Relationships**
  - 1:1
  - 1:N
  - M:N

**Inheritance**

- Interface (is-a) inheritance using "::"
- Class inheritance using `extends`
A graphical ODB schema for UNIVERSITY database

- **Person**
  - Work_in: Faculty
  - Has_majors: Department

- **Faculty**
  - advises: Student
  - In_committee_of: Committee

- **Student**
  - Major_in: Department
  - Registered_in: Section

- **Department**
  - offers: Course

- **Course**
  - Has_sections: Section
  - Offered_by: Faculty

- **Section**
  - students: Student
  - Of_course: CurrSection

- **GradStudent**
  - advisor: Faculty

- **Committee**
  - advisor: Faculty
  - registered_students: CurrSection
ODL Examples (1): A Very Simple Class

class Degree {
    attribute string college;
    attribute string degree;
    attribute string year;
};

(all examples are based on the university schema presented in Chapter 4 and graphically shown on page 680):
A class definition with “extent”, “key”, and more elaborate attributes; still relatively straightforward

```c
class Person (extent persons key ssn) {
    attribute struct Pname {string fname ...} name;
    attribute string ssn;
    attribute date birthdate;
    ...
    short age();
}
```
ODL Examples (3): A Class With Relationships

- Note extends (inheritance) relationship

- Also note “inverse” relationship

Class Faculty extends Person (extent faculty) {
    attribute string rank;
    attribute float salary;
    attribute string phone;
    ...
    relationship Dept works_in inverse Dept::has_faculty;
    relationship set<GradStu> advises inverse GradStu::advisor;
    void give_raise (in float raise);
    void promote (in string new_rank);
};
interface GeometryObject
{
    attribute enum Shape{Rectangle, Triangle, Circle,...} shape;
    attribute struct Point {short x, short y} reference_point;
    float perimeter();
    float area();
    void translate(in short x_translation; in short y_translation);
    void rotate(in float angle_of_rotation);
};

only operations are inherited, not properties as a result noninstantiable
Inheritance via “:” – An Example

interface GeometryObject {
    attribute struct point {...} reference_point;
    float perimeter();
    ...
};

class Triangle : GeometryObject (extent triangles) {
    attribute short side_1;
    attribute short side_2;
    ...
};
Object Query Language

- OQL is DMG’s query language
- OQL works closely with programming languages such as C++
- Embedded OQL statements return objects that are compatible with the type system of the host language
- OQL’s syntax is similar to SQL with additional features for objects
Object Query Language (OQL)

- **basic OQL syntax**
  - `select ... from ... where ...`
  - Retrieve the names of all departments in the college of ‘Engineering’

Q0: `SELECT d.dname
    FROM d in departments
    WHERE d.college = ‘Engineering’;`

How to refer to a persistent object?
Entry point (named persistent object; or name of the extent of a class)
Data Type of Query Results

- The data type of a query result can be any type defined in the ODMG model.

- A query does not have to follow the `select...from...where...` format.

- A persistent name on its own can serve as a query whose result is a reference to the persistent object.

  - Example

    `departments;` whose output is `set<Departments>`
Path Expressions

- A *path expression* is used to specify a path to attributes and objects in an entry point.
- A path expression starts at a persistent object name (or its iterator variable).
- The name will be followed by zero or more dot connected relationship or attribute names, e.g.,

  `departments.chair;`
Views as Named Objects

- The *define* keyword in OQL is used to specify an identifier for a *named query*.

- The name should be unique; if not, the results will replace an existing named query.

- Once a query definition is created, it will persist until deleted or redefined.

- A view definition can include parameters.
An Example of OQL View

- A view to include students in a department who have a minor:

  ```
  define has_minor(dept_name) as
  select s
  from s in students
  where s.minor_in.dname=dept_name
  ```

- `has_minor` can now be used in queries.
Single Elements from Collections

- An OQL query returns a collection

- OQL’s `element` operator can be used to return a single element from a singleton collection that contains one element:

  ```
  element (select d
          from d in departments
          where d.dname = 'Software Engineering');
  ```

- If `d` is empty or has more than one elements, an `exception` is raised
Collection Operators

- OQL supports a number of aggregate operators that can be applied to query results

- The aggregate operators include min, max, count, sum, and avg and operate over a collection

- count returns an integer; others return the same type as the collection type
An Example of an OQL: Aggregate Operator

- To compute the average GPA of all seniors majoring in Business:

```oql
avg ( select s.gpa
    from s in students
    where s.class = 'senior'
    and s.majors_in.dname = 'Business');
```
Membership and Quantification

- OQL provides membership and quantification operators:
  - \((e \text{ in } c)\) is true if \(e\) is in the collection \(c\)
  - \((\text{for all } e \text{ in } c : b)\) is true if all \(e\) elements of collection \(c\) satisfy \(b\)
  - \((\text{exists } e \text{ in } c : b)\) is true if at least one \(e\) in collection \(c\) satisfies \(b\)
An Example of Membership

- To retrieve the names of all students who completed ICS102:

```sql
select s.name.fname s.name.lname
from s in students
where 'ICS102' in
  (select c.name
   from c in s.completed_sections.section.of_course);
```
Ordered Collections

- Collections that are lists or arrays allow retrieving their first, last, and ith elements.
- OQL provides additional operators for extracting a sub-collection and concatenating two lists.
- OQL also provides operators for ordering the results.
An Example of Ordered Operation

To retrieve the last name of the faculty member who earns the highest salary:

```csharp
first (select struct
(faculty: f.name.lastname, salary f.salary)
from f in faculty
ordered by f.salary desc);
```
Grouping Operator

- OQL also supports a grouping operator called \texttt{group by}.

- To retrieve average GPA of majors in each department having >100 majors:

\begin{verbatim}
select deptname, avg_gpa:
  avg (select p.s.gpa from p in partition)
from s in students
group by deptname: s.majors_in.dname
having count (partition) > 100
\end{verbatim}
Object Database Conceptual Design

- Object Database (ODB) vs Relational Database (RDB)
  - Relationships are handled differently
  - Inheritance is handled differently
  - Operations in OBD are expressed early on since they are a part of the class specification
Relationships: ODB vs RDB (1)

- Relationships in ODB:
  - relationships are handled by reference attributes that include OIDs of related objects
  - single and collection of references are allowed
  - references for binary relationships can be expressed in single direction or both directions via inverse operator
Relationships in RDB:

- Relationships among tuples are specified by attributes with matching values (via *foreign keys*)
- Foreign keys are single-valued
- $M:N$ relationships must be presented via a separate relation (table)
Inheritance Relationship in ODB vs RDB

- Inheritance structures are built in ODB (and achieved via “:” and extends operators)

- RDB has no built-in support for inheritance relationships; there are several options for mapping inheritance relationships in an RDB (see Chapter 7)
Early Specification of Operations

- Another major difference between ODB and RDB is the specification of operations
  - ODB: operations specified during design (as part of class specification)
  - RDB: may be delayed until implementation
Mapping EER Schemas to ODB Schemas

- Mapping EER schemas into ODB schemas is relatively simple especially since ODB schemas provide support for inheritance relationships

- Once mapping has been completed, operations must be added to ODB schemas since EER schemas do not include an specification of operations
Mapping EER to ODB Schemas

Step 1:

- Create an ODL class for each EER entity type or subclass
  - Multi-valued attributes are declared by sets, bags or lists constructors
  - Composite attributes are mapped into tuple constructors
Mapping EER to ODB Schemas

Step 2:

- Add relationship properties or reference attributes for each binary relationship into the ODL classes participating in the relationship

  - Relationship cardinality: single-valued for 1:1 and N:1 directions; set-valued for 1:N and M:N directions

  - Relationship attributes: create via tuple constructors
Mapping EER to ODB Schemas

Step 3:

- Add appropriate operations for each class
  - Operations are not available from the EER schemas; original requirements must be reviewed
  - Corresponding *constructor* and *destructor* operations must also be added
Step 4:

- Specify inheritance relationships via `extends` clause

- An ODL class that corresponds to a sub-class in the EER schema inherits the types and methods of its super-class in the ODL schemas

- Other attributes of a sub-class are added by following Steps 1-3
Mapping EER to ODB Schemas

- **Step 5:**
  - Map weak entity types in the same way as regular entities
    - Weak entities that do not participate in any relationships may alternatively be presented as *composite multi-valued attribute* of the owner entity type
Mapping EER to ODB Schemas

Step 6:

- Map categories (union types) to ODL
  - The process is not straightforward
  - May follow the same mapping used for EER-to-relational mapping:
    - Declare a class to represent the category
    - Define 1:1 relationships between the category and each of its super-classes
Mapping EER to ODB Schemas

Step 7:

- Map $n$-ary relationships whose degree is greater than 2
  - Each relationship is mapped into a separate class with appropriate reference to each participating class
END